Chapter 3 – Classification

This notebook contains all the sample code and solutions to the exercises in chapter 3.

Open in Colab | Open in Kaggle

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20.

```
[ ] # Python ≥3.5 is required
    import sys
    assert sys.version_info >= (3, 5)

    # Is this notebook running on Colab or Kaggle?
    IS_COLAB = "google.colab" in sys.modules
    IS_KAGGLE = "kaggle_secrets" in sys.modules

    # Scikit-Learn ≥0.20 is required
    import sklearn
    assert sklearn.__version__ >= "0.20"

    # Common imports
    import numpy as np
    import os

    # to make this notebook's output stable across runs
    np.random.seed(42)

    # To plot pretty figures
    %matplotlib inline
    import matplotlib as mpl
    import matplotlib.pyplot as plt
    mpl.rc('axes', labelsize=14)
```

```python
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "classification"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

MNIST

Warning: since Scikit-Learn 0.24, `fetch_openml()` returns a Pandas `DataFrame` by default. To avoid this and keep the same code as in the book, we use `as_frame=False`.

```python
[ ] # get the name and version
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
mnist.keys()
```

```
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCI
```

```python
[ ] # count the attributs
X, y = mnist["data"], mnist["target"]
X.shape
```

```
(70000, 784)
```

```
[ ] #pick an element
    y[0]
```

```
'5'
```

```
[ ] y = y.astype(np.uint8)
```

```
[ ] def plot_digit(data):
        image = data.reshape(28, 28)
        plt.imshow(image, cmap = mpl.cm.binary,
                   interpolation="nearest")
        plt.axis("off")
```

```
[ ] # EXTRA
    def plot_digits(instances, images_per_row=10, **options):
        size = 28
        images_per_row = min(len(instances), images_per_row)
        images = [instance.reshape(size,size) for instance in instances]
        n_rows = (len(instances) - 1) // images_per_row + 1
        row_images = []
        n_empty = n_rows * images_per_row - len(instances)
        images.append(np.zeros((size, size * n_empty)))
        for row in range(n_rows):
            rimages = images[row * images_per_row : (row + 1) * images_per_row]
            row_images.append(np.concatenate(rimages, axis=1))
        image = np.concatenate(row_images, axis=0)
        plt.imshow(image, cmap = mpl.cm.binary, **options)
        plt.axis("off")
```

```
# print the first 100 digits in setting format
plt.figure(figsize=(9,9))
example_images = X[:100]
plot_digits(example_images, images_per_row=10)
save_fig("more_digits_plot")
plt.show()
```

```
Saving figure more_digits_plot
```

```
[ ] y.shape
```

```
(70000,)
```

```
[ ] 28 * 28
```

```
784
```

```
[ ] # import python plotting library
    %matplotlib inline
    import matplotlib as mpl
    import matplotlib.pyplot as plt

    #select an element
    some_digit = X[0]
    some_digit_image = some_digit.reshape(28, 28)
    plt.imshow(some_digit_image, cmap=mpl.cm.binary)
    plt.axis("off")

    save_fig("some_digit_plot")
    plt.show()
```

```
Saving figure some_digit_plot
```

```
[ ] #pick an element
    y[0]
```

```
y[0]
```

'5'

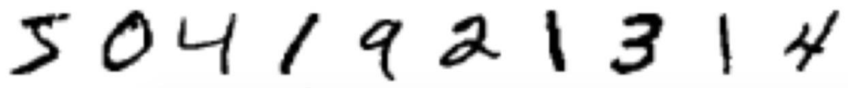
```
[ ] y = y.astype(np.uint8)
```

```
[ ] def plot_digit(data):
        image = data.reshape(28, 28)
        plt.imshow(image, cmap = mpl.cm.binary,
                   interpolation="nearest")
        plt.axis("off")
```

```
[ ] # EXTRA
    def plot_digits(instances, images_per_row=10, **options):
        size = 28
        images_per_row = min(len(instances), images_per_row)
        images = [instance.reshape(size,size) for instance in instances]
        n_rows = (len(instances) - 1) // images_per_row + 1
        row_images = []
        n_empty = n_rows * images_per_row - len(instances)
        images.append(np.zeros((size, size * n_empty)))
        for row in range(n_rows):
            rimages = images[row * images_per_row : (row + 1) * images_per_row]
            row_images.append(np.concatenate(rimages, axis=1))
        image = np.concatenate(row_images, axis=0)
        plt.imshow(image, cmap = mpl.cm.binary, **options)
        plt.axis("off")
```

```
# print the first 100 digits in setting format
plt.figure(figsize=(9,9))
example_images = X[:100]
plot_digits(example_images, images_per_row=10)
save_fig("more_digits_plot")
plt.show()
```

Saving figure more_digits_plot



```
Saving figure more_digits_plot
```

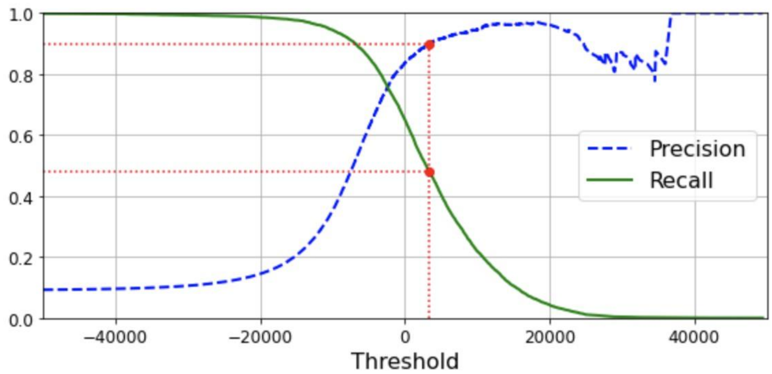
```
[ ] def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
        plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
        plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
        plt.legend(loc="center right", fontsize=16) # Not shown in the book
        plt.xlabel("Threshold", fontsize=16)        # Not shown
        plt.grid(True)                              # Not shown
        plt.axis([-50000, 50000, 0, 1])             # Not shown



    recall_90_precision = recalls[np.argmax(precisions >= 0.90)]
    threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]


    plt.figure(figsize=(8, 4))
    plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
    plt.plot([threshold_90_precision, threshold_90_precision], [0., 0.9], "r:")
    plt.plot([-50000, threshold_90_precision], [0.9, 0.9], "r:")
    plt.plot([-50000, threshold_90_precision], [recall_90_precision, recall_90_precision], "r:
    plt.plot([threshold_90_precision], [0.9], "ro")
    plt.plot([threshold_90_precision], [recall_90_precision], "ro")
    save_fig("precision_recall_vs_threshold_plot")
    plt.show()
```

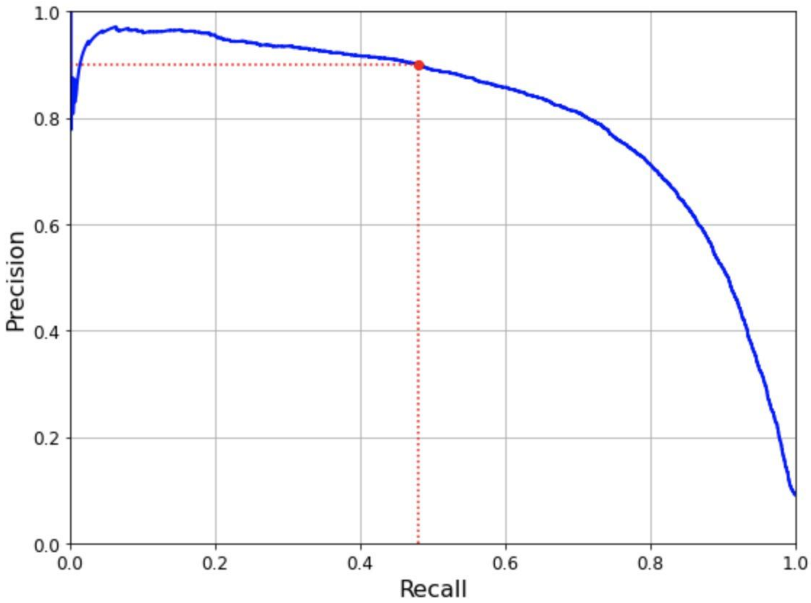
```
Saving figure precision_recall_vs_threshold_plot
```



```
[ ]  def plot_precision_vs_recall(precisions, recalls):
         plt.plot(recalls, precisions, "b-", linewidth=2)
         plt.xlabel("Recall", fontsize=16)
         plt.ylabel("Precision", fontsize=16)
         plt.axis([0, 1, 0, 1])
         plt.grid(True)

     plt.figure(figsize=(8, 6))
     plot_precision_vs_recall(precisions, recalls)
     plt.plot([recall_90_precision, recall_90_precision], [0., 0.9], "r:")
     plt.plot([0.0, recall_90_precision], [0.9, 0.9], "r:")
     plt.plot([recall_90_precision], [0.9], "ro")
     save_fig("precision_vs_recall_plot")
     plt.show()
```

```
Saving figure precision_vs_recall_plot
```

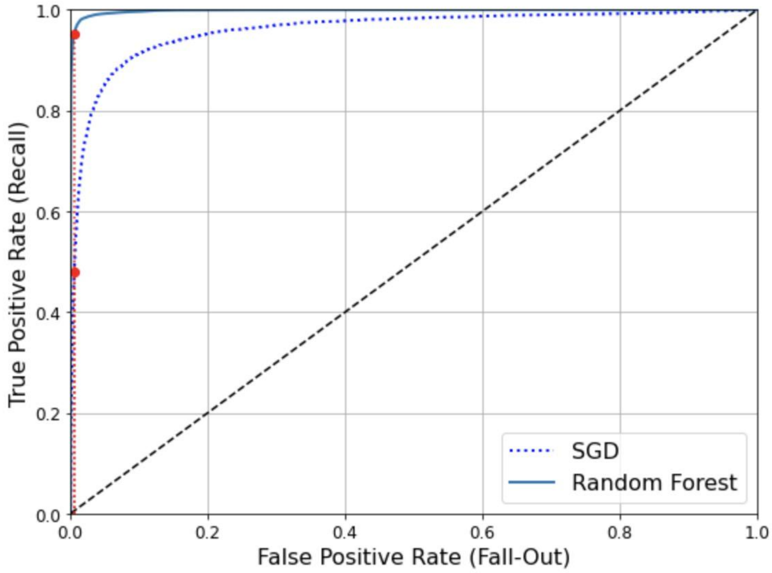



```
[ ]  threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
```

```
[ ] recall_for_forest = tpr_forest[np.argmax(fpr_forest >= fpr_90)]

    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, "b:", linewidth=2, label="SGD")
    plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
    plt.plot([fpr_90, fpr_90], [0., recall_90_precision], "r:")
    plt.plot([0.0, fpr_90], [recall_90_precision, recall_90_precision], "r:")
    plt.plot([fpr_90], [recall_90_precision], "ro")
    plt.plot([fpr_90, fpr_90], [0., recall_for_forest], "r:")
    plt.plot([fpr_90], [recall_for_forest], "ro")
    plt.grid(True)
    plt.legend(loc="lower right", fontsize=16)
    save_fig("roc_curve_comparison_plot")
    plt.show()
```

```
Saving figure roc_curve_comparison_plot
```



```python
from sklearn.pipeline import Pipeline

preprocess_pipeline = Pipeline([
    ("email_to_wordcount", EmailToWordCounterTransformer()),
    ("wordcount_to_vector", WordCounterToVectorTransformer()),
])

X_train_transformed = preprocess_pipeline.fit_transform(X_train)
```

Note: to be future-proof, we set `solver="lbfgs"` since this will be the default value in Scikit-Learn 0.22.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

log_clf = LogisticRegression(solver="lbfgs", max_iter=1000, random_state=42)
score = cross_val_score(log_clf, X_train_transformed, y_train, cv=3, verbose=3)
score.mean()
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.1s remaining:    0.0s
```

```
[CV]  ................................................................
[CV] .................................... , score=0.981, total=   0.1s
[CV]  ................................................................
[CV] .................................... , score=0.985, total=   0.2s
[CV]  ................................................................
[CV] .................................... , score=0.991, total=   0.2s
```

```
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.3s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.5s finished
```

```
0.9858333333333333
```

Over 98.5%, not bad for a first try! :) However, remember that we are using the "easy" dataset. You can try with the harder datasets, the results won't be so amazing. You would have to try multiple models, select the best ones and fine-tune them using cross-validation, and so on.

But you get the picture, so let's stop now, and just print out the precision/recall we get on the test set:

```python
from sklearn.metrics import precision_score, recall_score

X_test_transformed = preprocess_pipeline.transform(X_test)

log_clf = LogisticRegression(solver="lbfgs", max_iter=1000, random_state=42)
log_clf.fit(X_train_transformed, y_train)

y_pred = log_clf.predict(X_test_transformed)

print("Precision: {:.2f}%".format(100 * precision_score(y_test, y_pred)))
print("Recall: {:.2f}%".format(100 * recall_score(y_test, y_pred)))
```

```
Precision: 95.88%
Recall: 97.89%
```